Amendments to the Specification:

Please amend the first full paragraph on page 9, from lines 4 – 8 on page 9, as follows:

Figure 1B is a diagram illustrating accessibility of various elements in the operating system ~~10~~ and the processor according to one embodiment of the invention. For illustration purposes, only elements of ring-0 10 and ring-3 40 are shown. The various elements in the logical operating architecture 50 access an accessible physical memory 60 according to their ring hierarchy and the execution mode.

Please amend the paragraph that begins at line 25 of page 17 and extends through line 25 of page 18 as follows:

The processor nub 18 includes a master binding key (BK0) 202. The BK0 202 is generated at random when the processor nub 18 is first invoked, i.e., when it is first executed on the secure platform 200. The key generator 240 generates a key operating system nub key (OSNK) 203 to be used by the usage protector 250. The key generator 240 receives the OS Nub identifier 201 and the BK0 202 to generate an OSNK 203. There are a number of ways for the key generator 240 to generate the OSNK 203. The key generator 240 generates the OSNK 203 by combining the BK0 202 and the OS Nub ID 201 using a cryptographic hash function. In one embodiment, the OS nub ID 201 identifies the OS nub 16 being installed into the secure platform 200. The OS nub ID 201 can be the hash of the OS nub 16, or a hash of a certificate that authenticates the OS nub 16, or an ID value extracted from a certificate that authenticates the OS nub 16. It is noted that a cryptographic hash function is a one-way function, mathematical or otherwise, which takes a variable-length input string, called a pre-image and converts it

to a fixed-length, generally smaller, output string referred to as a hash value. The hash function is one-way in that it is difficult to generate a pre-image that matches the hash value of another pre-image. In one embodiment, the OS nub ID 201 is a hash value of one of the OS Nub 16 and a certificate representing the OS nub 16. Since the security of an algorithm rests in the key, it is important to choose a strong cryptographic process when generating a key. The software environment 210 may include a plurality of subsets (e.g., subset 230). The usage of the software environment 210 or the usage of the subset 230 is protected by the usage protector 250. The usage protector 250 uses the OSNK 203 to protect the usage of the subset 230. The software environment 210 may include an operating system (e.g., a ~~Windows~~ WINDOWS® operating system, a ~~Windows~~ WINDOWS® 95 operating system, a ~~Windows~~ WINDOWS® 98 operating system, a ~~Windows~~ WINDOWS NT® operating system, ~~Windows~~ WINDOWS® 2000 operating system) or a data base. The subset 230 may be a registry in the ~~Windows~~ WINDOWS® operating system or a subset of a database. Elements can be implemented in hardware or software.

Please amend the fourth paragraph of page 19, from lines 20 – 22 of page 19, and the fifth paragraph , which begins at the bottom of page 19, at line 23, and goes through the top of page 20, through line 9, as follows:

Figure 3A is a diagram illustrating the usage protector 250 shown in Figure 2 according to one embodiment of the invention. The usage protector 250 includes a compressor 310, ~~and~~ an encryptor 360, and a storage 398.

In one embodiment, the compressor 310 receives the subset 230 and compresses the subset 230 to generate a compressed subset 395. The encryptor 360 then encrypts the compressed subset 395 using the OSNK 203. The OSNK 203 is provided to the usage

protection 250 by the key generator 240 as shown in Figure 2. The compressed subset 395 is applied in the encryption process to save time (i.e. speed up) in the encryption process and/or space for storing the subset 230 in a memory. In another embodiment, the encryptor 360 takes the subset 230 without going through the compressing process and encrypts the subset 230 to generate an encrypted subset using the OSNK 203. The encrypting of the compressed subset or the subset 230 prevents unauthorized reads of the subset 230. To establish an authorized read, a request can be make to the OS nub and if the request is granted, the OSNK 203 is used to decrypt the encrypted or compressed encrypted subset 230.

Please amend the long paragraph beginning at line 3 of page 22 and extending to line 15 of page 23 as follows:

The decryptor 345 accepts the OS nub's encrypted private key 204, and decrypts it using the OSNK 203, exposing the private key 348 for use in the isolated environment. The manifest generator 335 generates a manifest 307 for the subset 230. The manifest 307 represents the subset 230 in a concise manner. The manifest 307 may include a number of descriptors or entities, which characterize some relevant aspects of the subset 230. Typically, these relevant aspects are particular or specific to the subset 230 so that two different subsets have different manifests. In one embodiment, the manifest 307 represents a plurality of entities (i.e., a collection of entities) where each entry in the manifest 307 represents a hash (e.g., unique fingerprint) over one entity in the collection. The subset 230 is partitioned into one or more group where each group has a pointer and associated hash in the manifest 307. The manifest 307 is stored in a storage medium 349 for later use. The manifest 307 is also input to the signature generator 340 to generate a signature 308 over the manifest using the private key 348. The generated signature 308 is also stored in a storage medium 349. At a later time, we desire to verify that the portions

of the subset 230 described by the manifest have not changed. This requires verifying that the manifest itself has not been changed, and that each group in subset 230 described by the manifest has not been changed. the stored signature and manifest are retrieved from the storage medium 349. The retrieved signature 309, and the retrieved manifest 354, along with the public key 205, are used by the signature verifier 350 to test that the retrieved manifest 354 is unchanged from the original manifest 307. The signature verifier 350 produces a signature-verifier flag 351, which is asserted when the signature verifies that the manifest is unchanged. In one embodiment, the verification process includes decrypting the retrieved signature 309 using the public key 205 to expose the before hash value. The retrieved manifest 354 is hashed to generate an after hash value. The before hash value is compared to the after hash value to detect whether the retrieved manifest 354 has been modified. If the two hash values match, the retrieved manifest 354 is the same as it was when the signature was generated. The retrieved manifest 354 is also supplied to the manifest verifier 355, which uses the descriptive information in the retrieved manifest 354 to selectively verify portions of subset 230. In a typical embodiment, this involves hashing each group in subset 230, where the group is identified by information in the retrieved manifest 354, and comparing the newly generated hash value against the hash value for the group stored in the retrieved manifest 354. The manifest verifier 355 produces a manifest-verified flag 356, which is asserted if all entries described by the retrieved manifest 354 are verified as unchanged. If a test center 357 determines that both the manifest-verified flag and the signature-verified flag are asserted, then the overall verification process passes, and selected portions of subset 230 are known to be the same as when the signed manifest was originally generated, and a pass/fail flag 358 is asserted. Note that the signature verifier 350 and the manifest verifier 355 can be invoked in any order.

Please amend the paragraph beginning at line 20 of page 23 and extending through line 19 of page 24 as follows:

The first encryptor 305 encrypts the first hash value 206 using the OSNK 203. The first hash value 206 is provided by the hashing function 220 as shown in Figure 2. The first encryptor 305 takes the first hash value 206 and encrypts it to generate an encrypted first hash value 302 using the OSNK. The encrypted hash value 302 is then stored in a storage 310 for later use. The encryption by the OSNK 203 allows the encrypted hash value to be stored in arbitrary (i.e., unprotected) storage media. Storage medium 310 may be any type of medium capable of storing information (e.g., the encrypted hash value 302). The storage medium 310 may be any type of disk, i.e., floppy disks, hard disks and optical disks) or any type of tape, i.e., tapes. The second encryptor 365 takes the second hash value 312 to encrypts it to generate an encrypted second hash value 301 using the OSNK 203. The second hash value 312 is provided by the hash function 220. The first encryptor 305 and the second encryptor 365 uses the same encryption algorithm, and this algorithm produces identical repeatable results for a given input. The encrypted first hash value 302 is now retrieved from the storage 310 for comparing with the second encrypted hash value 301. The comparator 315 compares the encrypted second hash value 301 with the retrieved encrypted first hash value 303 to detect if the subset 230 has been modified or tampered with. In the case where the subset 230 is deliberately updated by an authorized agent, the stored encrypted hash value is also updated. Since the modification of the subset 230 is authorized, the second encrypted hash value 301 is the same as the retrieved first encrypted first hash value 303. In the case where the subset 230 has been unauthorized modified or tampered with, the comparator 315 generates a modified/not-modified flag 341 indicating the subset has been modified and therefore, the subset 230 should not be used. An attacker cannot simply replace the first encrypted hash value 303 with one corresponding to the

unauthorized modified subset 230, because the attacker does not have access to the OSNK 203 used to encrypt the hash value.

Please amend the second full paragraph on page 26, from lines 5 – 16 on page 26, as follows:

Upon START, the process 700 obtains the OSNK and a hash value (Block 701). Next, the process 700 determines whether this is the first hash value (Block 702). If this is the first time the subset is encrypted, the obtained hash value is a first hash value. The process 700 encrypts the first hash value (Block 703) using the OSNK and stores the encrypted first hash value in a storage (Block 704). The process 800 is terminated. If, the process 700 determines that it is not the first hash value then the obtained hash value is a second hash value, the process 700 then encrypts the second hash value using the OSNK (Block 705). The process 700 then retrieves the encrypted first hash value from the storage (Block 706). Next, the process 700 compares the retrieved encrypted first hash value and the encrypted second hash value (Block 707). From the result of the comparison, the process 700 then generates a modified/not-modified flag (Block 708) after ~~determine~~ determining whether the subset has been modified. The process 700 is terminated.